Table of Contents

Process Pinning

Process/Thread Pinning Overview

Columbia Phase Out:

As of Feb. 8, 2013, the Columbia21 node has been taken offline as part of the Columbia phase out process. Columbia22-24 are still available. If your script requires a specific node, please make the appropriate changes in order to ensure the success of your job.

Summary: Pinning, the binding of a process or thread to a specific core, can improve the performance of your code by increasing the percentage of local memory accesses.

Once your code runs and produces correct results on a system, the next concern is its performance. For a code that uses multiple cores, the placement of processes and/or threads can play a significant role in code performance.

Given a set of processor cores in a PBS job, the Linux kernel usually does a reasonably good job of mapping processes/threads to physical cores (although the kernel may also migrate processes/threads). Some OpenMP runtime libraries and MPI libraries may also perform certain placements by default. In cases where the placements by the kernel or the MPI or OpenMP libraries are suboptimal, you can try multiple methods to control the placement in order to improve performance of your code. Using the same placement also has the added benefit of reducing runtime variability from run to run.

You should pay attention to maximizing data locality while minimizing latency and resource contention, and should have a clear understanding of the characteristics of your own code and the machine that the code is running on.

Characteristics of NAS HECC Systems

Pleiades and Columbia are two distinctly different types of systems.

Pleiades

Pleiades is a cluster system consisting of four different processor types -- Harpertown, Nehalem, Westmere, and Sandy Bridge, with a total of 11,776 nodes. On Pleiades, memory on each node is accessible and shared only by processes/threads running on that node.

A Harpertown node is a symmetric memory system where all 8 cores have equal access to the memory on the node, so data locality is not an issue.

On the other hand, a Nehalem-EP, Westmere, or Sandy Bridge node contains two sockets. Within each socket is a symmetric memory system. Accessing memory across the two sockets is through the Quick Path Interconnect and these nodes are considered non-uniform memory access (NUMA) systems. So,for optimal performance, data locality should not be overlooked on these three processor types.

Overall, compared to a global shared-memory NUMA system such as Columbia, data locality is less of a concern on Pleiades. Rather, minimizing latency and resource contention will be the main focus when pinning processes/threads on Pleiades.

For more information on Pleiades and these processors, see Pleiades Configuration Details, which has links to each of the processor types.

Columbia

Columbia comprises 4 hosts (C*21-24*). Each host is a NUMA system that contains hundreds of nodes with memory located physically at various distances from the processors accessing data on memory. A process/thread can access the local memory on its node, as well as the remote memory across nodes through the NUMAlink, with varying latencies. So, data locality is critical for getting good performance on Columbia.

One good practice to follow when developing an application is to initialize data in parallel, such that each processor core initializes data that it is likely to access later for calculation.

For more information about Columbia, see Columbia Configuration Details.

Methods for Process/Thread Pinning

Several pinning approaches for OpenMP, MPI and MPI+OpenMP hybrid applications are listed below. We recommend using the Intel compiler (and its runtime library) and the SGI MPT software on NAS systems, so most of the approaches pertain specifically to them. On the other hand, the **`mbind`** tool works for multiple OpenMP libraries and MPI environments.

- OpenMP codes
    - ♦ Using Intel OpenMP Thread Affinity for Pinning
    - ♦ Using SGI's omplace Tool for Pinning
    - ♦ Using the mbind Tool for Pinning
- MPI codes

    - Setting SGI MPT Environment Variables
    - Using SGI's omplace Tool for Pinniing
    - Using the mbind Tool for Pinning
- MPI+OpenMP hybrid codes
    - Using SGI's omplace Tool for Pinning
    - Using the mbind Tool for Pinning

Checking Process/Thread Placement

Each of the approaches listed above provides some verbose capability to print out the tool's placement results. In addition, you can check the placement using the following approaches:

ps Command

`ps -C` *`executable_name`* `-L -opsr,comm,time,pid,ppid,lwp`
In the output generated, use the core ID under the PSR column, the process ID under the PID column, and the thread ID under the LWP column to see where the processes and/or threads are placed on the cores.

Note that the **`ps`** command provides a snapshot of the placement at that specific time. You may need to monitor the placement from time to time to make sure that the processes/threads do not migrate.

Instrument your code to

- Call the **`mpi_get_processor_name`** function, to get the name of the processor an MPI process is running on
- Call the Linux C function **`sched_getcpu()`** to get the processor number the process or thread is running on

For more information, see Instrumenting your Fortran Code to Check Process/Thread Placement.

Using SGI's dplace Tool for Pinning

Summary: The **dplace** tool binds processes/threads to specific processor cores to improve your code performance. For an introduction to pinning at NAS, see Process/Thread Pinning Overview.

The SGI **dplace** tool binds processes/threads to specific processor cores. Once pinned, the processes/threads do not migrate. This can improve the performance of your code by increasing the percentage of local memory accesses.

dplace invokes a kernel module to create a job placement container consisting of all (or a subset of) the CPUs of the cpuset. In the current **dplace** version 2, an LD_PRELOAD library (**libdplace.so**) is used to intercept calls to the functions **fork()**, **exec()**, and **pthread_create()** to place tasks that are being created. Note that tasks created internal to **glib** are not intercepted by the preload library. These tasks will *not* be placed. If no placement file is being used, then the **dplace** process is placed in the job placement container and (by default) is bound to the first CPU of the cpuset associated with the container.

Syntax

```
dplace [-e] [-c cpu_numbers] [-s skip_count] [-n process_name] \
             [-x skip_mask] [-r [l|b|t]] [-o log_file] [-v 1|2] \
             command [command-args]
dplace [-p placement_file] [-o log_file] command [mpiexec -np4 a.out]
dplace [-q] [-qq] [-qqq]
```

As illustrated above, **dplace** "execs" **command** (in this case, without its **mpiexec** arguments), which executes within this placement container and continues to be bound to the first CPU of the container. As the command forks child processes, they inherit the container and are bound to the next available CPU of the container.

If a placement file is being used, then the **dplace** process is not placed at the time the job placement container is created. Instead, placement occurs as processes are forked and executed.

Options for dplace

Explanations for some of the options are provided below. For additional information, see **man dplace** on either Pleiades or Columbia.

-e and -c *cpu_numbers*

-e determines exact placement. As processes are created, they are bound to CPUs in the exact order specified in the CPU list. CPU numbers may appear multiple times in the list.

A CPU value of "**x**" indicates that binding should *not* be done for that process. If the end of the list is reached, binding starts over again at the beginning of the list.

-c *cpu_numbers* specifies a list of CPUs, optionally strided CPU ranges, or a striding pattern. For example:

- -c 1
- -c 2-4 (equivalent to -c 2,3,4)
- -c 12-8 (equivalent to -c 12,11,10,9,8)
- -c 1,4-8,3
- -c 2-8:3 (equivalent to -c 2,5,8)
- -c CS
- -c BT

NOTE: CPU numbers are *not* physical CPU numbers. They are logical CPU numbers that are relative to the CPUs that are in the allowed set, as specified by the current cpuset.

A CPU value of "**x**" (or *****), in the argument list for the **-c** option, indicates that binding should not be done for that process. The value "**x**" should be used only if the **-e** option is also used.

Note that CPU numbers start at 0.

For striding patterns, any subset of the characters (**B**)lade, (**S**)ocket, (**C**)ore, (**T**)hread may be used; their ordering specifies the nesting of the iteration. For example, **SC** means to iterate all the cores in a socket before moving to the next CPU socket, while **CB** means to pin to the first core of each blade, then the second core of every blade, and so on.

For best results, use the **-e** option when using stride patterns. If the **-c** option is not specified, all CPUs of the current cpuset are available. The command itself (which is "execed" by **dplace**) is the first process to be placed by the **-c *cpu_numbers***.

Without the **-e** option, the order of numbers for the **-c** option is not important.

-x *skip_mask*

Provides the ability to skip placement of processes. The ***skip_mask*** argument is a bitmask. If bit *N* of ***skip_mask*** is set, then the *N*+1th process that is forked is not placed. For example, setting the mask to 6 prevents the second and third processes from being

placed. The first process (the process named by the command) will be assigned to the first CPU. The second and third processes are not placed. The fourth process is assigned to the second CPU, and so on. This option is useful for certain classes of threaded applications that spawn a few helper processes that typically do not use much CPU time.

-s *skip_count*

Skips the first ***skip_count*** processes before starting to place processes onto CPUs. This option is useful if the first ***skip_count*** processes are "shepherd" processes used only for launching the application. If ***skip_count*** is not specified, a default value of 0 is used.

-q

Lists the global count of the number of active processes that have been placed (by **dplace**) on each CPU in the current cpuset. Note that CPU numbers are logical CPU numbers within the cpuset, not physical CPU numbers. If specified twice, lists the current **dplace** jobs that are running. If specified three times, lists the current **dplace** jobs and the tasks that are in each job.

-o *log_file*

Writes a trace file to ***log_file*** that describes the placement actions that were made for each fork, exec, etc. Each line contains a time-stamp, process id:thread number, CPU that task was executing on, taskname and placement action. Works with version 2 only.

Examples of dplace Usage

For OpenMP Codes

```
#PBS -lselect=1:ncpus=8

#With Intel compiler versions 10.1.015 and later,
#you need to set KMP_AFFINITY to disabled
#to avoid the interference between dplace and
#Intel's thread affinity interface.

setenv KMP_AFFINITY disabled

#The -x2 option provides a skip map of 010 (binary 2) to
#specify that the 2nd thread should not be bound. This is
#because under the new kernels (including the ones used on
#Pleiades and Columbia), the master thread (first thread)
#will fork off one monitor thread (2nd thread) which does
#not need to be pinned.
```

```
#On Pleiades, if the number of threads is less than
#the number of cores, choose how you want
#to place the threads carefully. For example,
#the following placement is good on Harpertown
#but not good on other Pleiades processor types:

dplace -x2 -c 2,1,4,5 ./a.out
```

To check the thread placement, you can add the **-o** option to create a log:

```
dplace -x2 -c 2,1,4,5 -o log_file ./a.out
```

Or use the following command on the running host while the job is still running:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp > placement.out
```

Sample Output of log_file

```
timestamp       process:thread cpu taskname| placement action
15:32:42.196786 31044          1 dplace    | exec ./openmp1, ncpu 1
15:32:42.210628 31044:0        1 a.out     | load, cpu 1
15:32:42.211785 31044:0        1 a.out     | pthread_create thread_number 1, ncpu -1
15:32:42.211850 31044:1        - a.out     | new_thread
15:32:42.212223 31044:0        1 a.out     | pthread_create thread_number 2, ncpu 2
15:32:42.212298 31044:2        2 a.out     | new_thread
15:32:42.212630 31044:0        1 a.out     | pthread_create thread_number 3, ncpu 4
15:32:42.212717 31044:3        4 a.out     | new_thread
15:32:42.213082 31044:0        1 a.out     | pthread_create thread_number 4, ncpu 5
15:32:42.213167 31044:4        5 a.out     | new_thread
15:32:54.709509 31044:0        1 a.out     | exit
```

Sample Output of placement.out

```
PSR COMMAND             TIME   PID  PPID   LWP
  1 a.out           00:00:02 31044 31039 31044
  0 a.out           00:00:00 31044 31039 31046
  2 a.out           00:00:02 31044 31039 31047
  4 a.out           00:00:01 31044 31039 31048
  5 a.out           00:00:01 31044 31039 31049
```

Note that Intel OpenMP jobs use an extra thread that is unknown to the user and it does not need to be placed. In the above example, this extra thread (31046) is running on core number 0.

For MPI Codes Built with SGI's MPT Library

With SGI's MPT, only 1 shepherd process is created for the entire pool of MPI processes, and the proper way of pinning using **dplace** is to skip the shepherd process.

Here is an example for Columbia:

```
#PBS -l ncpus=8
....
 mpirun -np 8 dplace -s1 -c 0-7 ./a.out
# or
 mpiexec -np 8 dplace -s1 -c 0-7 ./a.out
```

On Pleiades, if the number of processes in each node is less than the number of cores in that node, choose how you want to place the processes carefully. For example, the following placement works well on Harpertown nodes, but not on other Pleiades processor types:

```
#PBS -l select=2:ncpus=8:mpiprocs=4
...
mpiexec -np 8 dplace -s1 -c 2,4,1,5 ./a.out
```

To check the placement, you can set MPI_DSM_VERBOSE, which prints the placement in the PBS **stderr** file:

```
#PBS -l select=2:ncpus=8:mpiprocs=4
...
setenv MPI_DSM_VERBOSE
mpiexec -np 8 dplace -s1 -c 2,4,1,5 ./a.out
```

Output in PBS stderr File

```
MPI: DSM information
grank   lrank   pinning  node name      cpuid
    0       0   yes      r75i2n13           1
    1       1   yes      r75i2n13           2
    2       2   yes      r75i2n13           4
    3       3   yes      r75i2n13           5
    4       0   yes      r87i2n6            1
    5       1   yes      r87i2n6            2
    6       2   yes      r87i2n6            4
    7       3   yes      r87i2n6            5
```

If you use the **-o *log_file*** flag of **dplace**, the CPUs where the processes/threads are placed will be printed, but the node names are not printed.

```
#PBS -l select=2:ncpus=8:mpiprocs=4
....
mpiexec -np 8 dplace -s1 -c 2,4,1,5 -o log_file ./a.out
```

Output in log_file

```
timestamp       process:thread cpu taskname | placement action
15:16:35.848646 19807          - dplace     | exec ./new_pi_mpt126, ncpu -1
15:16:35.877584 19807:0        - a.out      | load, cpu -1
15:16:35.878256 19807:0        - a.out      | fork -> pid 19810, ncpu 1
15:16:35.879496 19807:0        - a.out      | fork -> pid 19811, ncpu 2
15:16:35.880053 22665:0        - a.out      | fork -> pid 22672, ncpu 2
15:16:35.880628 19807:0        - a.out      | fork -> pid 19812, ncpu 4
15:16:35.881283 22665:0        - a.out      | fork -> pid 22673, ncpu 4
15:16:35.882536 22665:0        - a.out      | fork -> pid 22674, ncpu 5
15:16:35.881960 19807:0        - a.out      | fork -> pid 19813, ncpu 5
15:16:57.258113 19810:0        1 a.out      | exit
15:16:57.258116 19813:0        5 a.out      | exit
15:16:57.258215 19811:0        2 a.out      | exit
15:16:57.258272 19812:0        4 a.out      | exit
15:16:57.260458 22672:0        2 a.out      | exit
15:16:57.260601 22673:0        4 a.out      | exit
15:16:57.260680 22674:0        5 a.out      | exit
15:16:57.260675 22671:0        1 a.out      | exit
```

For MPI Codes Built with MVAPICH2 Library

With MVAPICH2, 1 shepherd process is created for each MPI process. You can use **`ps -L -u your_userid`** on the running node to see these processes. To properly pin MPI processes using **`dplace`**, you cannot skip the shepherd processes and must use the following:

```
mpiexec -np 4 dplace -c2,4,1,5 ./a.out
```

Using Intel OpenMP Thread Affinity for Pinning

Columbia Phase Out:

As of Feb. 8, 2013, the Columbia21 node has been taken offline as part of the Columbia phase out process. Columbia22-24 are still available. If your script requires a specific node, please make the appropriate changes in order to ensure the success of your job.

Summary: The Intel compiler's OpenMP runtime library has the ability to bind OpenMP threads to physical processing units. Depending on the system (machine) topology, application, and operating system, thread affinity can have a dramatic effect on the code performance. For most OpenMP codes, **`type=scatter`** would provide the best performance, as it minimizes cache and memory bandwidth contention for Nehalem-EP, Westmere, and Sandy Bridge. For Harpertown, using an explicit **`proclist`** should give the best performance.

Recommended Approaches

Two approaches are recommended for using the Intel OpenMP thread affinity capability:

Use the KMP_AFFINITY Environment Variable

The thread affinity interface is controlled using the KMP_AFFINITY environment variable.

Syntax

For **`csh`** and **`tcsh`**:

```
setenv KMP_AFFINITY [<modifier>,...]<type>[,<permute>][,<offset>]
```

For **`sh, bash,`** and **`ksh`**:

```
export KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]
```

Use the Compiler Flag -par-affinity Compiler Option

Starting with the Intel compiler version 11.1, thread affinity can also be specified through the compiler option **`-par-affinity`**. The use of **`-openmp`** or **`-parallel`** is required in order for this option to take effect. This option overrides the environment variable when both are specified. See **man ifort** for more information.

Syntax

`-par-affinity=[<modifier>,...]<type>[,<permute>][,<offset>]`
For both of these approaches, **`type`** is the only required argument, and it indicates the type of thread affinity to use. Descriptions of the arguments (**`type`**, **`modifier`**, **`permute`**, and **`offset`**) can be found on Intel's Thread Affinity Interface web page.

Note: Intel compiler versions 11.1 and later are recommended, as some of the affinity methods described below are not supported in earlier versions.

Possible Values of type

Possible values for **`type`** are:

type = none (default)

Does not bind OpenMP threads to particular thread contexts; however, if the operating system supports affinity, the compiler still uses the OpenMP thread affinity interface to determine machine topology. Specify **`KMP_AFFINITY=verbose,none`** to list a machine topology map.

type = disabled

Specifying **`disabled`** completely disables the thread affinity interfaces. This forces the OpenMP runtime library to behave as if the affinity interface was not supported by the operating system. This includes implementations of the low-level API interfaces such as **`kmp_set_affinity`** and **`kmp_get_affinity`** that have no effect and will return a nonzero error code.

Additional information from Intel:

"The thread affinity type of KMP_AFFINITY environment variable defaults to none (**`KMP_AFFINITY=none`**). The behavior for **`KMP_AFFINITY=none`** was changed in 10.1.015 or later, and in all 11.*x* compilers, such that the initialization thread creates a "full mask" of all the threads on the machine, and every thread binds to this mask at startup time. It was subsequently found that this change may interfere with other platform affinity mechanism, for example, **`dplace()`** on SGI Altix machines. To resolve this issue, a new affinity type **`disabled`** was introduced in compiler 10.1.018, and in all 11.*x* compilers (**`KMP_AFFINITY=disabled`**). Setting **`KMP_AFFINITY=disabled`** will prevent the runtime library from making any affinity-related system calls."

type = compact

Specifying **`compact`** causes the threads to be placed as close together as possible. For example, in a topology map, the nearer a core is to the root, the more significance the core has when sorting the threads.

Usage example:

```
# for sh, ksh, bash
export KMP_AFFINITY=compact,verbose

# for csh, tcsh
setenv KMP_AFFINITY compact,verbose
```

type = scatter

Specifying **`scatter`** distributes the threads as evenly as possible across the entire system. Scatter is the opposite of compact.

Usage example:

```
# for sh, ksh, bash
export KMP_AFFINITY=scatter,verbose

# for csh, tcsh
setenv KMP_AFFINITY scatter,verbose
```

type = explicit

Specifying **`explicit`** assigns OpenMP threads to a list of OS proc IDs that have been explicitly specified by using the **`proclist=`** modifier, which is required for this affinity type.

Usage example:

```
# for sh, ksh, bash
export KMP_AFFINITY="explicit,proclist=[0,1,4,5],verbose"

# for csh, tcsh
setenv KMP_AFFINITY "explicit,proclist=[0,1,4,5],verbose"
```

For nodes that support hyper-threading (such as Nehalem-EP, Westmere, and Sandy Br), you can use the **`granularity`** modifier to choose whether to pin OpenMP threads to physical cores using **`granularity=core`** (the default) or pin to logical cores using

granularity=fine or **granularity=thread** for the **compact** and **scatter** types.

For most OpenMP codes, **type=scatter** should provide the best performance, as it minimizes cache and memory bandwidth contention for Nehalem-EP, Westmere, and Sandy Bridge nodes. For Harpertown nodes, using an explicit **proclist** should give the best performance.

Examples

The following examples illustrate the thread placement of an OpenMP job with four threads on various platforms with different thread affinity methods. The variable OMP_NUM_THREADS is set to 4:

```
# for sh, ksh, bash
export OMP_NUM_THREADS=4

# for csh, tcsh
setenv OMP_NUM_THREADS 4
```

The use of the **verbose** modifier is recommended, as it provides an output with the placement.

Harpertown

Note that every two cores (indicated with same color) in Harpertown share L2 cache.

Four threads running on one node (eight physical cores) of Harpertown will get the following thread placement:

setting of KMP_AFFINITY	Processor id	0	2	4	6	1	3	5	7
compact,verbose	thread id	0	1	2	3				
scatter,verbose	thread id	0	2			1	3		
"explicit,proclist=[0,1,4,5],verbose"	thread id	0		2		1		3	

Nehalem-EP

Note that four physical cores (indicated with same color) in Nehalem-EP share the same L3 cache.

Four threads running on one node (eight physical cores and 16 logical cores due to hyper-threading) of Nehalem-EP will get the following thread placement:

setting of KMP_AFFINITY	Processor id	0,8	1,9	2,10	3,11	4,12	5,13	6,14	7,15

granularity=core,compact,verbose	thread id	0,1	2,3								
granularity=core,scatter,verbose	thread id	0	2					1	3		
"explicit,proclist=[0,2,4,6],verbose"	thread id	0		1				2		3	

Note that with **granularity=core**, an OpenMP thread is pinned to a physical core, and is allowed to float between the two logical cores associated with the physical core. For example, with **granularity=core,compact**, both threads 0 and 1 are pinned to the logical core set {0,8}. If you use **granularity=fine,compact** instead, thread 0 is pinned to logical core 0 and thread 1 is pinned to logical core 8, respectively.

Westmere

Note that six physical cores (indicated with same color) in Westmere share the same L3 cache.

Four threads running on 1 node (12 physical cores and 24 logical cores due to hyper-threading) of Westmere will get the following thread placement:

setting of KMP_AFFINITY	Processor id	0,12	1,13	2,14	3,15	4,16	5,17	6,18	7,19	8,20	9,21	10,22	11
granularity=core,compact,verbose	thread id	0,1	2,3										
granularity=core,scatter,verbose	thread id	0	2					1	3				
"explicit,proclist=[0,3,6,9],verbose"	thread id	0			1			2			3		

Sandy Bridge

As seen in the configuration diagram of a Sandy Bridge node, each set of eight physical cores in a socket share the same L3 cache.

Four threads running on 1 node (16 physical cores and 32 logical cores due to hyper-threading) of Sandy Bridge will get the following thread placement:

setting of KMP_AFFINITY	Processor id	0,16	1,17	2,18	3,19	4,20	5,21
granularity=core,compact,verbose	thread id	0,1	2,3				
granularity=core,scatter,verbose	thread id	0	2				
"explicit,proclist=[0,4,8,12],verbose"	thread id	0				1	

Click Here to Expand Table

Columbia

Each Columbia host has hundreds of cores. Based on the number of cores requested by the PBS job, a cpuset is created with the requested number of cores. Depending on availability, PBS may not be able to allocate consecutive cores to a job.

There are two cores per node (indicated with same color, below) on Columbia21, while

there are four cores per node on C22-24. In the following example, 8 consecutive cores (cores 4-11) are allocated on Columbia21.

Four threads running on 8 cores of Columbia21 will get the following thread placement:

setting of KMP_AFFINITY	Processor id	4	5	6	7	8	9	10	11
compact,verbose	thread id	0	1	2	3				
scatter,verbose	thread id	0		1		2		3	
"explicit,proclist=[5,7,9,11],verbose"	thread id		0		1		2		3

Using SGI MPT Environment Variables for Pinning

Summary: For MPI codes built with SGI's MPT libraries, one way to control pinning is to set certain MPT memory placement environment variables. For an introduction to pinning at NAS, see Process/Thread Pinning Overview.

MPT Environment Variables

Here are the MPT memory placement environment variables:

MPI_DSM_VERBOSE

Directs MPI to display a synopsis of the NUMA and host placement options being used at run time to the standard error file.

Default: not enabled

The setting of this environment variable is ignored if **`MPI_DSM_OFF`** is also set.

MPI_DSM_DISTRIBUTE

Activates NUMA job placement mode. This mode ensures that each MPI process gets a unique CPU and physical memory on the node with which that CPU is associated. Currently, the CPUs are chosen by simply starting at relative CPU 0 and incrementing until all MPI processes have been forked.

SGI defaults:

- off for MPT.1.25
- on for MPT.1.26, MPT.2.0.1, MPT.2.0.4, MPT.2.0.6

NAS local defaults:

- off for PBS jobs using Harpertown nodes
- on for PBS jobs using Nehalem, Westmere, and Sandy Bridge nodes
- on for PBS jobs on Columbia

WARNING: Under most situations, it is a bad practice to set this environment variable for running on the Harpertown nodes. For the Nehalem and Westmere nodes, it is recommended that you do *not* set this environment variable if the nodes are not fully populated with MPI processes. This is because the CPUs are chosen sequentially from relative CPU 0.

The setting of this environment variable is ignored if **MPI_DSM_CPULIST** is also set or if **dplace** or **omplace** is used.

MPI_DSM_CPULIST

Specifies a list of CPUs on which to run an MPI application, excluding the shepherd process(es) and **mpirun**. The number of CPUs specified should equal the number of MPI processes (excluding the shepherd process) that will be used.

Syntax and examples for the list:

- Use a comma and/or hyphen to provide a delineated list:

```
# place MPI processes ranks 0-2 on CPUs 2-4
# and ranks 3-5 on CPUs 6-8
setenv MPI_DSM_CPULIST "2-4,6-8"
```

- Use a "/" and a stride length to specify CPU striding:

```
# Place the MPI ranks 0 through 3 stridden
# on CPUs 8, 10, 12, and 14
setenv MPI_DSM_CPULIST 8-15/2
```

- Use a colon to separate CPU lists of multiple hosts:

```
# Place the MPI processes 0 through 7 on the first host
# on CPUs 8 through 15. Place MPI processes 8 through 15
# on CPUs 16 to 23 on the second host.
setenv MPI_DSM_CPULIST 8-15:16-23
```

- Use a colon followed by **allhosts** to indicate that the prior list pattern applies to all subsequent hosts/executables:

```
# Place the MPI processes onto CPUs 0, 2, 4, 6 on all hosts
setenv MPI_DSM_CPULIST 0-7/2:allhosts
```

Examples

An MPI job requesting 2 nodes on Pleiades and running 4 MPI processes per node will get the following placements, depending on the environment variables set:

```
#PBS -lselect=2:ncpus=8:mpiprocs=4
module load <a_mpt_module>
setenv ....
cd $PBS_O_WORKDIR
mpiexec -np 8 ./a.out
```

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_DISTRIBUTE

```
MPI: DSM information
MPI: MPI_DSM_DISTRIBUTE enabled
grank   lrank   pinning  node name      cpuid
    0       0   yes      r86i3n5            0
    1       1   yes      r86i3n5            1
    2       2   yes      r86i3n5            2
    3       3   yes      r86i3n5            3
    4       0   yes      r86i3n6            0
    5       1   yes      r86i3n6            1
    6       2   yes      r86i3n6            2
    7       3   yes      r86i3n6            3
```

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_CPULIST 0,2,4,6

```
MPI: WARNING MPI_DSM_CPULIST CPU placement spec list is too short.
MPI:         MPI processes on host #1 and later will not be pinned.
MPI: DSM information
grank   lrank   pinning  node name      cpuid
    0       0   yes      r22i1n7            0
    1       1   yes      r22i1n7            2
    2       2   yes      r22i1n7            4
    3       3   yes      r22i1n7            6
    4       0   no      r22i1n8 0
    5       1   no      r22i1n8 0
    6       2   no      r22i1n8 0
    7       3   no      r22i1n8 0
```

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_CPULIST 0,2,4,6:0,2,4,6

```
MPI: DSM information
grank   lrank   pinning  node name      cpuid
    0       0   yes      r13i2n12           0
    1       1   yes      r13i2n12           2
    2       2   yes      r13i2n12           4
    3       3   yes      r13i2n12           6
    4       0   yes      r13i3n7            0
    5       1   yes      r13i3n7            2
    6       2   yes      r13i3n7            4
    7       3   yes      r13i3n7            6
```

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_CPULIST 0,2,4,6:allhosts

```
MPI: DSM information
grank   lrank   pinning  node name      cpuid
    0       0   yes      r13i2n12           0
    1       1   yes      r13i2n12           2
    2       2   yes      r13i2n12           4
    3       3   yes      r13i2n12           6
    4       0   yes      r13i3n7            0
    5       1   yes      r13i3n7            2
    6       2   yes      r13i3n7            4
    7       3   yes      r13i3n7            6
```

Using SGI omplace for Pinning

Summary: The **omplace** wrapper script pins processes and threads for better performance. It works with SGI MPT, Intel MPI, and hybrid MPI/OpenMP applications.

SGI's **omplace** is a wrapper script for **dplace**. It provides an easier syntax than **dplace** for pinning processes and threads. **omplace** works with SGI MPT as well as with Intel MPI. In addition to pinning pure MPI or pure OpenMP applications, **omplace** can also be used for pinning hybrid MPI/OpenMP applications.

A few issues with **omplace** to keep in mind:

- **dplace** and **omplace** do not work with Intel compiler versions 10.1.015 and 10.1.017. Use the Intel compiler version 11.1 or later, instead
- To avoid interference between **dplace/omplace** and Intel's thread affinity interface, set the environment variable KMP_AFFINITY to **disabled** or set OMPLACE_AFFINITY_COMPAT to ON
- The **omplace** script is part of SGI's MPT, and is located under the **/nasa/sgi/mpt/*mpt_version_number*/bin** directory

Syntax

```
For OpenMP:
setenv OMP_NUM_THREADS nthreads
omplace [OPTIONS] program args...
or
omplace -nt nthreads [OPTIONS] program args...

For MPI:
mpiexec -np nranks omplace [OPTIONS] program args...

For MPI/OpenMP hybrid:
setenv OMP_NUM_THREADS nthreads
mpiexec -np nranks omplace [OPTIONS] program args...
or
mpiexec -np nranks omplace -nt nthreads [OPTIONS] program args...
```

Some useful **omplace** options are listed below:

-b *basecpu*
: Specifies the starting CPU number for the effective CPU list.

-c *cpulist*
: Specifies the effective CPU list. This is a comma-separated list of CPUs or CPU ranges.

  WARNING: For **omplace**, a blank space is required between **-c** and **cpulist**. Without the space, the job will fail. This is different from **dplace**.

-nt *nthreads*
: Specifies the number of threads per MPI process. If this option is unspecified, it defaults to the value set for the OMP_NUM_THREADS environment variable. If OMP_NUM_THREADS is not set, then *nthreads* defaults to 1.

-v
: Verbose option. Portions of the automatically generated placement file will be displayed.

-vv
: Very verbose option. The automatically generated placement file will be displayed in its entirety.

For information about additional options, see **man omplace**.

Examples

For Pure OpenMP Codes Using the Intel OpenMP Library

Sample PBS script:

```
#PBS -lselect=1:ncpus=12:model=wes

module load comp-intel/11.1.072
setenv KMP_AFFINITY disabled

omplace -c 0,3,6,9 -vv ./a.out
```

Sample placement information for this script is given in the application's **stout** file:

```
omplace: placement file /tmp/omplace.file.21891
    firsttask cpu=0
    thread oncpu=0 cpu=3-9:3 noplace=1  exact
```

The above placement output may not be easy to understand. A better way to check the placement is to run the **ps** command on the running host while the job is still running:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp > placement.out
```

Sample output of **placement.out**

```
PSR COMMAND             TIME   PID  PPID   LWP
  0 openmp1         00:00:02 31918 31855 31918
 19 openmp1         00:00:00 31918 31855 31919
  3 openmp1         00:00:02 31918 31855 31920
  6 openmp1         00:00:02 31918 31855 31921
  9 openmp1         00:00:02 31918 31855 31922
```

Note that Intel OpenMP jobs use an extra thread that is unknown to the user, and does not need to be placed. In the above example, this extra thread is running on logical core number 19.

For Pure MPI Codes Using SGI MPT

Sample PBS script:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes

module load comp-intel/11.1.072
module load mpi-sgi/mpt.2.04.10789

#Setting MPI_DSM_VERBOSE allows the placement information
#to be printed to the PBS stderr file

setenv MPI_DSM_VERBOSE

mpiexec -np 8 omplace -c 0,3,6,9 ./a.out
```

Sample placement information for this script is shown in the PBS **`stderr`** file:

```
MPI: DSM information
MPI: using dplace
grank   lrank   pinning  node name      cpuid
    0       0   yes      r144i3n12          0
    1       1   yes      r144i3n12          3
    2       2   yes      r144i3n12          6
    3       3   yes      r144i3n12          9
    4       0   yes      r145i2n3           0
    5       1   yes      r145i2n3           3
    6       2   yes      r145i2n3           6
    7       3   yes      r145i2n3           9
```

In this example, the four processes on each node are evenly distributed to the two sockets (CPUs 0 and 3 are on the first socket while CPUs 6 and 9 on the second socket) to minimize contention. If **`omplace`** had not been used, then placement would follow the rules of the environment variable **`OMP_DSM_DISTRIBUTE`**, and all four processes would have been placed on the first socket -- likely leading to more contention.

For MPI/OpenMP Hybrid Codes Using SGI MPT and Intel OpenMP

Proper placement is more critical for MPI/OpenMP hybrid codes than for pure MPI or pure OpenMP codes. The following example demonstrates the situation when no placement instruction is provided and the OpenMP threads for each MPI process are stepping on one another which likely would lead to very bad performance.

Sample PBS script without pinning:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes

module load comp-intel/11.1.072
module load mpi-sgi/mpt.2.04.10789
```

```
setenv OMP_NUM_THREADS 2

mpiexec -np 8 ./a.out
```

There are two problems with the resulting placement shown in the example above. First, you can see that the first four MPI processes on each node are placed on four cores (0,1,2,3) of the same socket, which will likely lead to more contention compared to when they are distributed between the two sockets.

```
MPI: MPI_DSM_DISTRIBUTE enabled
grank   lrank   pinning  node name      cpuid
    0       0   yes      r212i0n10          0
    1       1   yes      r212i0n10          1
    2       2   yes      r212i0n10          2
    3       3   yes      r212i0n10          3
    4       0   yes      r212i0n11          0
    5       1   yes      r212i0n11          1
    6       2   yes      r212i0n11          2
    7       3   yes      r212i0n11          3
```

The second problem is that, as demonstrated with the **ps** command below, the OpenMP threads are also placed on the same core where the associated MPI process is running:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp
PSR COMMAND         TIME   PID  PPID   LWP
  0 a.out       00:00:02  4098  4092  4098
  0 a.out       00:00:02  4098  4092  4108
  0 a.out       00:00:02  4098  4092  4110
  1 a.out       00:00:03  4099  4092  4099
  1 a.out       00:00:03  4099  4092  4106
  2 a.out       00:00:03  4100  4092  4100
  2 a.out       00:00:03  4100  4092  4109
  3 a.out       00:00:03  4101  4092  4101
  3 a.out       00:00:03  4101  4092  4107
```

Sample PBS script demonstrating proper placement:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes

module load mpi-sgi/mpt.2.04.10789
module load comp-intel/11.1.072

setenv MPI_DSM_VERBOSE
setenv OMP_NUM_THREADS 2
setenv KMP_AFFINITY disabled

cd $PBS_O_WORKDIR

#the following two lines will result in identical placement

mpiexec -np 8 omplace -nt 2 -c 0,1,3,4,6,7,9,10 -vv ./a.out
#mpiexec -np 8 omplace -nt 2 -c 0-10:bs=2+st=3 -vv  ./a.out
```

Shown in the PBS **stderr** file, the 4 MPI processes on each node are properly distributed on the two sockets with processes 0 and 1 on CPUs 0 and 3 (first socket) and processes 2 and 3 on CPUs 6 and 9 (second socket).

```
MPI: DSM information
MPI: using dplace
grank   lrank   pinning  node name      cpuid
    0       0   yes      r212i0n10          0
    1       1   yes      r212i0n10          3
    2       2   yes      r212i0n10          6
    3       3   yes      r212i0n10          9
    4       0   yes      r212i0n11          0
    5       1   yes      r212i0n11          3
    6       2   yes      r212i0n11          6
    7       3   yes      r212i0n11          9
```

In the PBS **stout** file, it shows the placement of the two OpenMP threads for each MPI process:

```
omplace: This is an SGI MPI program.
omplace: placement file /tmp/omplace.file.6454
    fork skip=0  exact cpu=0-10:3
    thread oncpu=0 cpu=1 noplace=1  exact
    thread oncpu=3 cpu=4 noplace=1  exact
    thread oncpu=6 cpu=7 noplace=1  exact
    thread oncpu=9 cpu=10 noplace=1  exact
omplace: This is an SGI MPI program.
omplace: placement file /tmp/omplace.file.22771
    fork skip=0  exact cpu=0-10:3
    thread oncpu=0 cpu=1 noplace=1  exact
    thread oncpu=3 cpu=4 noplace=1  exact
    thread oncpu=6 cpu=7 noplace=1  exact
    thread oncpu=9 cpu=10 noplace=1  exact
```

To get a better picture of how the OpenMP threads are placed, using the following **ps** command:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp
PSR COMMAND         TIME   PID  PPID   LWP
  0 a.out       00:00:06  4436  4435  4436
  1 a.out       00:00:03  4436  4435  4447
  1 a.out       00:00:03  4436  4435  4448
  3 a.out       00:00:06  4437  4435  4437
  4 a.out       00:00:05  4437  4435  4446
  6 a.out       00:00:06  4438  4435  4438
  7 a.out       00:00:05  4438  4435  4444
  9 a.out       00:00:06  4439  4435  4439
 10 a.out       00:00:05  4439  4435  4445
```

Using the mbind Tool for Pinning

Summary: The **mbind** utility is a "one-stop" tool for binding processes and threads for MPI and OpenMP, and hybrid applications.

The **mbind** utility, developed at NAS, is used for binding processes and threads to CPUs. It works for MPI, OpenMP, or MPI+OpenMP hybrid applications, and is available under **/u/scicon/tools/bin** on Pleiades.

One of the benefits of **mbind** is that it relieves users from the burden of learning the complexity of each individual pinning approach for associated MPI or OpenMP libraries. It providing a uniform usage model that works for multiple MPI and OpenMP environments.

Currently supported MPI and OpenMP libraries are listed below.

MPI:

- SGI-MPT
- MVAPICH2
- INTEL-MPI
- OPEN-MPI
- MPICH2

OpenMP:

- Intel OpenMP runtime library
- GNU OpenMP library
- PGI runtime library
- Pathscale OpenMP runtime library

Use of **mbind** is limited to cases where the same set of CPU lists is used for all processor nodes, and the same number of threads is used for all processes.

WARNING: Be aware that **mbind** may have issues when used together with other performance tools, such as PerfSuite.

Syntax

```
#For OpenMP:
mbind.x [-options] program [args]

#For MPI or MPI+OpenMP hybrid which supports mpiexec:
mpiexec -np nranks mbind.x [-options] program [args]
```

Information about all available options can be found in the text file **/u/scicon/tools/doc/mbind.txt** on Pleiades.

Here are a few recommended **mbind** options:

-cs, -cp, -cc; or -c*cpulist*
: **-cs** for spread (default), **-cp** for compact, **-cc** for cyclic; **-ccpulist** for process ranks (for example, **-c*0,3,6,9***). CPU numbers in the *cpulist* are relative within a cpuset if present.
Note that the **-cs** option will spread the processes and threads among the physical cores to minimize various resource contentions, and is usually the best choice for placement.

-n*n*
: Number of processes per node.

-t*n*
: Number of threads per process. The default value is given by the OMP_NUM_THREADS environment variable.

-v*n*
: Verbose flag; print some useful information. [*n*] controls the level of details. Default is **n=0** (OFF).

Examples

For Pure OpenMP Codes Using Intel OpenMP Library

Sample PBS script:

```
#PBS -l select=1:ncpus=12:model=wes
#PBS -l walltime=0:5:0

module load comp-intel/11.1.072

cd $PBS_O_WORKDIR

mbind.x -cs -t4 -v ./a.out
```

The 4 OpenMP threads are spread (with the **-cs** option) among 4 physical cores in a node, as shown in the application's **stdout**:

```
host: r211i0n5, ncpus 24, nthreads: 4, bound to cpus: {0,3,6,9}
```

The proper placement is further demonstrated in the output of the **ps** command below:

```
r211i0n5% ps -C a.out -L -opsr,comm,time,pid,ppid,lwp
PSR COMMAND             TIME   PID  PPID   LWP
  9 a.out           00:02:06   849   711   849
  3 a.out           00:00:00   849   711   850
  3 a.out           00:02:34   849   711   851
  0 a.out           00:01:47   849   711   852
  6 a.out           00:01:23   849   711   853
```

Note that Intel OpenMP creates an extra thread, which is unknown to the user and does not need to be placed. In this example, this extra thread (thread id 850) is running on the same core (core 3) as thread 851. Since this extra thread does not do any work, it will not interfere with the other threads.

For Pure MPI Codes Using SGI MPT

WARNING: **`mbind.x`** overwrites the placement initially performed by MPT's **`mpiexec`**. The placement output from MPI_DSM_VERBOSE (if set) most likely is incorrect and should be ignored.
Sample PBS script:

```
#PBS -l select=1:ncpus=12:model=wes

module load comp-intel/11.1.072
module load mpi-sgi/mpt.2.04.10789

#setenv MPI_DSM_VERBOSE

cd $PBS_O_WORKDIR

mpiexec -np 8 mbind.x -cs -n4 -v ./a.out
```

On each of the two nodes, 4 MPI processes are spread among 4 physical cores (0,3,6,9), as shown in the application's **`stdout`**:

```
host: r213i2n12, ncpus 24, process-rank: 0, bound to cpu: 0
host: r213i2n12, ncpus 24, process-rank: 1, bound to cpu: 3
host: r213i2n12, ncpus 24, process-rank: 3, bound to cpu: 9
host: r213i2n12, ncpus 24, process-rank: 2, bound to cpu: 6
host: r213i2n13, ncpus 24, process-rank: 4, bound to cpu: 0
host: r213i2n13, ncpus 24, process-rank: 5, bound to cpu: 3
host: r213i2n13, ncpus 24, process-rank: 6, bound to cpu: 6
host: r213i2n13, ncpus 24, process-rank: 7, bound to cpu: 9
```

For MPI+OpenMP Hybrid Codes Using SGI MPT and Intel OpenMP

Sample PBS script:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes

module load comp-intel/11.1.072
module load mpi-sgi/mpt.2.04.10789

#setenv MPI_DSM_VERBOSE

cd $PBS_O_WORKDIR

mpiexec -np 8 mbind.x -cs -n4 -t2 -v ./a.out
```

On each of the two nodes, the 4 MPI processes are spread among the physical cores. The 2 OpenMP threads of each MPI process run on adjacent physical cores as seen in the application's **stdout**:

```
host: r215i2n12, ncpus 24, process-rank: 4, nthreads: 2, bound to cpus: {0-1}
host: r215i2n12, ncpus 24, process-rank: 6, nthreads: 2, bound to cpus: {6-7}
host: r215i2n12, ncpus 24, process-rank: 5, nthreads: 2, bound to cpus: {2-3}
host: r215i2n12, ncpus 24, process-rank: 7, nthreads: 2, bound to cpus: {8-9}
host: r215i2n11, ncpus 24, process-rank: 0, nthreads: 2, bound to cpus: {0-1}
host: r215i2n11, ncpus 24, process-rank: 2, nthreads: 2, bound to cpus: {6-7}
host: r215i2n11, ncpus 24, process-rank: 3, nthreads: 2, bound to cpus: {8-9}
host: r215i2n11, ncpus 24, process-rank: 1, nthreads: 2, bound to cpus: {2-3}
```

For MPI+OpenMP Hybrid Codes Using MVAPICH2 and Intel OpenMP

Sample PBS script:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes

module load comp-intel/11.1.072
module load mpi-mvapich2/1.4.1/intel

cd $PBS_O_WORKDIR

mpiexec -np 8 mbind.x -cs -n4 -t2 -v ./a.out

#If you use mpirun_rsh instead of mpiexec
#use the following

mpirun_rsh -np 8 -hostfile $PBS_NODEFILE \
mbind.x -cs -n4 -t2 -v2 ./a.out
```

The application's **stdout** in this example is very similar to that in the previous MPT/Intel OpenMP example.

For MPI+OpenMP Hybrid Codes Using Intel MPI and Intel OpenMP

The Intel MPI library automatically pins processes to CPUs to prevent unwanted process migration. If you find that the placement done by the Intel MPI library is not optimal, you can use **mbind** to do the pinning instead.

WARNING: Note that in order for **mbind** to work with the Intel MPI library, the internal pinning mode of the library must be turned off explicitly by setting the environment variable I_MPI_PIN to *0*.
Sample PBS script:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes
```

```
module load comp-intel/11.1.072
module load mpi-intel/4.0.2.003

setenv I_MPI_PIN 0

cd $PBS_O_WORKDIR

mpdboot --file=$PBS_NODEFILE --ncpus=1 --totalnum=`cat $PBS_NODEFILE | \
 sort -u | wc -l` --ifhn=`head -1 $PBS_NODEFILE` --rsh=ssh \
 --mpd=`which mpd` --ordered

mpiexec -ppn 4 -np 8 mbind.x -cs -n4 -t2 -v ./a.out

mpdallexit
```

For the above job, the following placement is seen in the application's **stdout**:

```
host: r215i2n11, ncpus 24, process-rank: 0, nthreads: 2, bound to cpus: {0-1}
host: r215i2n11, ncpus 24, process-rank: 1, nthreads: 2, bound to cpus: {2-3}
host: r215i2n11, ncpus 24, process-rank: 3, nthreads: 2, bound to cpus: {8-9}
host: r215i2n11, ncpus 24, process-rank: 2, nthreads: 2, bound to cpus: {6-7}
host: r215i2n12, ncpus 24, process-rank: 5, nthreads: 2, bound to cpus: {2-3}
host: r215i2n12, ncpus 24, process-rank: 4, nthreads: 2, bound to cpus: {0-1}
host: r215i2n12, ncpus 24, process-rank: 7, nthreads: 2, bound to cpus: {8-9}
host: r215i2n12, ncpus 24, process-rank: 6, nthreads: 2, bound to cpus: {6-7}
```

This can be confirmed by running the following **ps** command on the running nodes. For clarity, the extra OpenMP threads created by the Intel OpenMP (which don't do any work) are removed from the output.

```
r215i2n11% ps -C hybrid_intelmpi -L -opsr,comm,time,pid,ppid,lwp
PSR COMMAND      TIME   PID  PPID   LWP
  6 a.out    00:00:12 44698 44696 44698
  7 a.out    00:00:12 44698 44696 44715
  2 a.out    00:00:12 44699 44695 44699
  3 a.out    00:00:12 44699 44695 44711
  8 a.out    00:00:12 44700 44697 44700
  9 a.out    00:00:12 44700 44697 44713
  0 a.out    00:00:12 44701 44694 44701
  1 a.out    00:00:12 44701 44694 44717
```

If I_MPI_PIN is not set to *0* in the PBS script, then **mbind.x** prints out identical placement results, as in the case where I_MPI_PIN is set to *0* but the **ps** command shows that some OpenMP threads "step on" one another.

```
r215i2n11% ps -C hybrid_intelmpi -L -opsr,comm,time,pid,ppid,lwp
PSR COMMAND      TIME   PID  PPID   LWP
  3 a.out    00:00:12 44185 44182 44185
  3 a.out    00:00:12 44185 44182 44198
  6 a.out    00:00:19 44186 44183 44186
  7 a.out    00:00:12 44186 44183 44202
  9 a.out    00:00:12 44187 44184 44187
  9 a.out    00:00:12 44187 44184 44200
```

```
  0 a.out     00:00:19 44188 44181 44188
  1 a.out     00:00:12 44188 44181 44204
```

The **mbind** utility was created by NAS staff member Henry Jin.

Instrumenting your Fortran Code to Check Process/Thread Placement

Summary: Pinning, the binding of a process or thread to a specific core, can improve the performance of your code. To check whether your Fortran code has been successfully pinned, use the C code, **`mycpu.c`**, found below.

The MPI function **`mpi_get_processor_name`** and the Linux C function **`sched_getcpu`** can be inserted into your source code to check process and/or thread placement.

The MPI function **`mpi_get_processor_name`** returns the hostname an MPI process is running on (to be used for MPI and/or MPI+OpenMP codes only). The Linux C function **`sched_getcpu`** returns the processor number the process/thread is running on.

If your source code is written in Fortran, you can use the C code, **`mycpu.c`**, below, which allows your Fortran code to call **`sched_getcpu`**.

C Program mycpu.c

```
#include <utmpx.h>
int sched_getcpu();

int findmycpu_ ()
{
    int cpu;
    cpu = sched_getcpu();
    return cpu;
}
```

Compile **`mycpu.c`** as follows to produce the object file **`mycpu.o`**:

```
pfe20% module load comp-intel/2011.2
pfe20% icc -c mycpu.c
```

The example below demonstrates how to instrument an MPI+OpenMP source code with the above functions. The added lines are shown in red.

```
      program your_program
      use omp_lib
...
      integer :: resultlen, tn, cpu
      integer, external :: findmycpu
      character (len=8) :: name

      call mpi_init( ierr )
      call mpi_comm_rank( mpi_comm_world, rank, ierr )
      call mpi_comm_size( mpi_comm_world, numprocs, ierr )
```

```
      call mpi_get_processor_name(name, resultlen, ierr)
!$omp parallel

      tn = omp_get_thread_num()
      cpu = findmycpu()
      write (6,*) 'rank ', rank, ' thread ', tn,
     &  ' hostname ', name, ' cpu ', cpu
.....
!$omp end parallel
      call mpi_finalize(ierr)
      end
```

Compile your instrumented code as follows:

```
pfe20% module load comp-intel/2011.2
pfe20% module load mpi-sgi/mpt.2.06a67
pfe20% ifort -o a.out -openmp mycpu.o your_program.f -lmpi
```

Sample PBS script

The PBS script below shows an example for running the hybrid MPI+OPenMP code across two nodes, with 2 MPI processes per node and 4 OpenMP threads per process, and using mbind to pin the processes and threads.

```
#PBS -lselect=2:ncpus=12:mpiprocs=2:model=wes
#PBS -lwalltime=0:10:00

cd $PBS_O_WORKDIR

module load comp-intel/2011.2
module load mpi-sgi/mpt.2.06a67

mpiexec -np 4 mbind.x -cs -n2 -t4 -v ./a.out
```

Here is a sample output:

```
These 4 lines are generated by mbind only if you have included the -v option:
host: r212i1n8, ncpus 24, process-rank: 0, nthreads: 4, bound to cpus: {0-3}
host: r212i1n8, ncpus 24, process-rank: 1, nthreads: 4, bound to cpus: {6-9}
host: r212i1n9, ncpus 24, process-rank: 2, nthreads: 4, bound to cpus: {0-3}
host: r212i1n9, ncpus 24, process-rank: 3, nthreads: 4, bound to cpus: {6-9}

These lines are generated by your instrumented code:
rank    0 thread    0 hostname r212i1n8 cpu    0
rank    0 thread    1 hostname r212i1n8 cpu    1
rank    0 thread    2 hostname r212i1n8 cpu    2
rank    0 thread    3 hostname r212i1n8 cpu    3
rank    1 thread    0 hostname r212i1n8 cpu    6
rank    1 thread    1 hostname r212i1n8 cpu    7
rank    1 thread    2 hostname r212i1n8 cpu    8
rank    1 thread    3 hostname r212i1n8 cpu    9
rank    2 thread    0 hostname r212i1n9 cpu    0
```

```
rank    2 thread    1 hostname r212i1n9 cpu    1
rank    2 thread    2 hostname r212i1n9 cpu    2
rank    2 thread    3 hostname r212i1n9 cpu    3
rank    3 thread    0 hostname r212i1n9 cpu    6
rank    3 thread    1 hostname r212i1n9 cpu    7
rank    3 thread    2 hostname r212i1n9 cpu    8
rank    3 thread    3 hostname r212i1n9 cpu    9
```

Note that these lines in your output may be listed in a different order.

This approach was suggested by NAS SGI analyst Ken Taylor.